PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization*

Mehmet E. Belviranli
Computer Science Dept.
Univ. of California, Riverside
belviram@cs.ucr.edu

Peng Deng
Elec. and Comp. Eng. Dept.
Univ. of California, Riverside
pdeng002@ucr.edu

Laxmi N. Bhuyan
Computer Science Dept.
Univ. of California, Riverside
bhuyan@cs.ucr.edu

Rajiv Gupta
Computer Science Dept.
Univ. of California, Riverside
gupta@cs.ucr.edu

Qi Zhu
Elec. and Comp. Eng. Dept.
Univ. of California, Riverside
qzhu@ece.ucr.edu

ABSTRACT

Nested loops with regular iteration dependencies span a large class of applications ranging from string matching to linear system solvers. Wavefront parallelism is a well-known technique to enable concurrent processing of such applications and is widely being used on GPUs to benefit from their massively parallel computing capabilities. Wavefront parallelism on GPUs uses global barriers between processing of tiles to enforce data dependencies. However, such diagonal-wide synchronization causes load imbalance by forcing SMs to wait for the completion of the SM with longest computation. Moreover, diagonal processing causes loss of locality due to elements that border adjacent tiles.

In this paper, we propose PeerWave, an alternative GPU wavefront parallelization technique that improves inter-SM load balance by using peer-wise synchronization between SMs. and eliminating global synchronization. Our approach also increases GPU L2 cache locality through row allocation of tiles to the SMs. We further improve PeerWave performance by using flexible hyper-tiles that reduce inter-SM wait time while maximizing intra-SM utilization. We develop an analytical model for determining the optimal tile size. Finally, we present a run-time and a CUDA based API to allow users to easily implement their applications using PeerWave. We evaluate PeerWave on the NVIDIA K40c GPU using 6 different applications and achieve speedups of up to 2X compared to the most recent hyperplane transformation based GPU implementation.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMD)*

General Terms

Algorithms

Keywords

Wavefront parallelism; GP-GPU computing; decentralized synchronization.

1. INTRODUCTION

Since nested loops are a major source of parallelism, a great deal of research has focused on parallel execution of nested loops. Many classes of applications like time-based simulations, linear system solvers, and string matching algorithms employ a combination of DOACROSS and DOALL parallelism to process large volumes of data efficiently on massively parallel systems. Application and architecture specific challenges imposed by the data dependencies between loop iterations have lead to the development of various compile-time [5, 11], run-time[2, 12, 9], and algorithm-level [15, 3, 13] solutions.

Wavefront parallelism [16] is a well known technique for exploiting parallelism in nested loops using multiple processing units. Based on the dependency relationships across iterations along space and/or time dimensions, the computation proceeds in form of *diagonal waves* where iterations in each wave can be executed in parallel. To enforce data dependencies across consecutive waves, their execution is serialized via the use of barriers. To exploit data locality, elements are grouped into square tiles [17] and each tile is assigned to a single processor. This approach introduces a second level of parallelism where tiles can also be processed in parallel along diagonal waves with global barriers separating them.

GPUs have been used to accelerate wavefront based applications via use of their massive number of processing units. Many frequently used algorithms such as Smith-Waterman [14, 18], Cholesky Factorization [15], and SOR loop nests [4] have been adapted to run efficiently on GPUs. The two level parallelism via tile partitioning is also commonly exploited by GPU based techniques whose evaluation has been reported in several studies [5, 4, 18]. In these approaches, a thread block (TB) is created for each tiles in a diagonal and these tiles are processed in parallel by the multiple streaming multipro-

*This work is supported by the National Science Foundation, under grant CCF-0905509, CNS-1157377 and CCF-1423108

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

DOI: http://dx.doi.org/10.1145/2751205.2751243.

cessors (SMs) of the GPU (*inter-tile* parallelism). Iterations along diagonals within a tile are also executed in parallel by the SIMD computation units (i.e. CUDA cores) inside an SM (*intra-tile* parallelism). Similar to CPU based approaches, global barriers are required for both tile-level (i.e. *inter-tile*) and element-level(i.e. *intra-tile*) diagonals in order to enforce data dependencies. However, on GPUs, using global barriers for 2-level tiled wavefront execution has two major drawbacks:

1. Low utilization due to load imbalance. The use of global barriers leads to low utilization of processing units during both intra- and inter-tile parallel execution. The intra-tile imbalance causes threads utilization to be lower towards the beginning and the end of diagonal iterations. The same thing happens in processing diagonals of tiles, where the number of tiles are less at the beginning and towards the end. Moreover, inter-tile processing is also imbalanced, when other processors wait for the longest running task (i.e., processing of a tile) in a diagonal. SMs are forced to stay idle, even though new tiles are ready to execute as their dependencies have been satisfied. Intra-tile load imbalance for GPUs is addressed in [5] where tiles are transformed into polyhedral planes [8] to maximize utilization of threads processing a tile. However, this approach still uses inter-tile global barriers causing some SMs to unnecessarily wait for the longest running task.

2. Loss of data locality across tiles is another cause of performance degradation in global barriers. In CUDA, thread blocks, hence tiles, are executed in no specific order; therefore there is no guarantee that neighboring tiles will be processed by the same SM. This behavior destroys the dependency-implied locality across the border elements of adjacent tiles. Efforts have been made to overcome this drawback of global barriers. To improve data locality, [10] statically assigns row of tiles to the same processor and for shared memory multi-processors large CPU caches significantly improve data reuse between neighboring tiles. However, intra-tile operations are still carried out on a diagonal basis, which constitute most of the computations. In contrast to CPUs, both intra-tile and inter-tile locality is not exploited on GPUs since L1/L2 caches are much smaller. The effects of row or column assignments of tiles on GPU execution has not been tried before.

In this paper we address the above issues by developing PeerWave, a new GPU parallelization scheme for nested loops with data dependencies. We develop a decentralized synchronization scheme called *PeerSM* which eliminates global barriers by utilizing efficient SM-to-SM communication. PeerSM considerably reduces load imbalance by decreasing redundant idle waits and letting SMs start executing tiles independently as soon as their dependencies are satisfied. Our approach also improves cache locality across consecutive tiles by *Row-to-SM* assignment. *Flexible Hyper-Tiles* further reduces SM under utilization by allowing fine-grained control over synchronization intervals while keeping intra-tile utilization at maximum.

Our paper has the following contributions:

- We design a fast SM-to-SM peer synchronization mechanism (*PeerSM*) to eliminate load imbalance between SMs.
- We implement a row allocation scheme (*Row-to-SM*) which assigns consecutive tiles to same SM via programming of thread blocks (TBs).
- We extend hyperplane tile transformation (hyper-tile) (*Flexible Hyper-Tiles*) to further decrease idleness via adjustable synchronization intervals .
- We evaluate our scheme on the state of the art NVIDIA K40c GPU for 6 different applications and achieve speedup of up to 2X compared to approach that uses global barriers and hyper-tiles.
- We develop a generic API for CUDA to support a wide class of applications with nested loops and regular dependencies.

The rest of this paper is organized as follows. In Section 2, we give background on wavefront parallelism and related techniques. In Section 3 we describe our scheme in detail and in Section 4&5 we elaborate on the details, optimizations, our run-time and API. In Section 6 we describe the applications we use and in Section 7, we present the evaluation of our scheme. We end the paper with discussion of related work and the conclusion.

2. WAVEFRONT PARALLELISM

The wavefront technique exploits parallelism in nested loops that contain regular data dependencies across loop iterations. The main idea is to *skew* the iteration space and re-order loops to obtain a DOALL parallelism in one of the loops [16]. An example loop nest that processes a 2D array with cross iteration dependencies is given in Algorithm 1. The original loop nest is serial, and by skewing the inner j loop, DOALL parallelism is obtained across the iterations of the outer i loop. To utilize this parallelism, the two loops are swapped and their boundaries are adjusted accordingly. Figure 1(a) illustrates the iterations of this skewed&swapped loop nest. The dependencies between elements are shown via arrows and *diagonal*s of elements are marked with dashed lines. Each diagonal corresponds to an iteration of the outer j loop that can be executed in parallel. An important property of such parallelism is its non-uniformity, where the parallelism first increases and then decreases along the outer diagonal iteration space.

Algorithm 1 Wavefront execution via loop skewing

```
// Original loop nest
for i=2 to n do
    for j=2 to m do
        A[i,j] = (A[i-1,j] + A[i,j-1])/2
    end for
end for

// Skewed
for i=2 to n do
    for j=i+2 to i+m do
        A[i,j-i] = (A[i-1,j-i] + A[i,j-i-1])/2
    end for
end for

// Skewed & Swapped
for j=4 to n+m do
    for i=MAX(2,j-m+2) to MIN(n,j-2) do
        A[i,j-i] = (A[i-1,j-i] + A[i,j-i-1])/2
    end for
end for
```

Tiling. In wavefront parallelism the control over granularity and locality is achieved by partitioning the iteration space into $t \times t$ square regions called *tiles* (see Figure 1(b)). Typically, tiling decreases inter-processor communication and also increases cache locality by grouping elements. Similar to the element-wise diagonal pattern, streaming multiprocessors

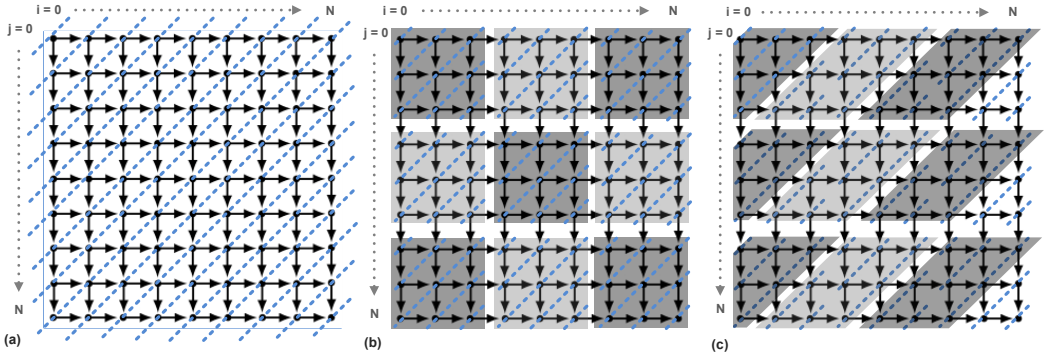

Figure 1: On the left(a), basic wavefront parallelism where dashed lines corresponds to 'waves'. In the center(b), tiling enables a second level of parallelism. On the right (c) tiles are transformed by the hyperplane technique.

(SMs) can be used to process diagonals of tiles in parallel (*inter-tile* parallelism) while diagonal elements in each tile can also be processed in parallel (*intra-tile* parallelism) by SIMD compute units (CUDA cores).

Even though GPUs enable 2-level parallelism, increased memory distance between neighboring diagonal executions combined with smaller cache sizes in GPUs reduces the possibility of cache hits and dramatically increases un-coalesced memory accesses. Moreover, due to the SIMD nature of GPGPU programming, any thread block (TB), hence a tile, can be executed on any SM, therefore voiding any possible spatial locality between neighboring tiles belonging to successive diagonals. The locality between two neighboring tiles in the same row is also lost because the data corresponding to only the last few diagonals is available in the cache. Hence straightforward allocation of two neighboring tiles to the same SM may not also benefit locality on GPUs. What we really need is the bordering data of a tile to be passed on to the same SM.

Hyperplane Transformation. An inherent drawback of using square/rectangular tiles in wavefront parallelism is the non-uniform diagonal sizes encountered during the execution of a tile. Hyperplane partitioning [8] converts square tiles into polyhedral quadrilaterals via a series of affine transformations. As shown in Figure 1(c), resulting tiles are composed of equal-sized diagonals with a total count of n, for n x n tiles whereas the square tiles end up in $2n-1$ diagonal iterations. Unlike square tiles, which limit maximum intra-tile parallelism to the minimum of tile width and height, hyper-plane partitioning allows arbitrary tile widths while keeping the diagonal size equal to the tile height. It may also be observed that the data is passed directly from the elements of a diagonal to be executed in the next iteration, hence increasing cache hits in the SM even for small cache sizes. If we allocate the next tile in the same row to the same SM, there will be cache hits for the bordering data. A technique similar to hyperplane partitioning for GPUs was proposed by [5] via compiler transformations to improve intra-tile parallelism. However, aforementioned inter-tile benefits have not been previously explored.

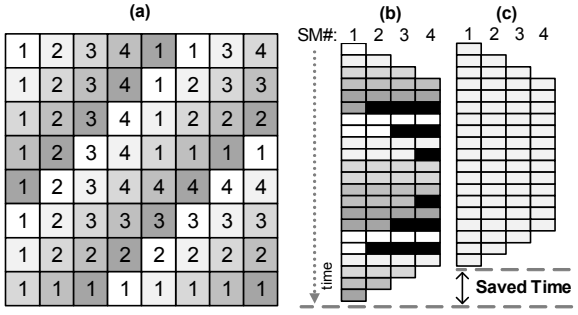

Figure 2: SM Under-utilization: On the left in (a) tiles are numbered with SMs executing them using global barriers; (b) shows the execution time-line of each SM using global barriers where dark-black slots indicate idle times; and (c) shows the ideal execution with the idle slots are eliminated.

Load Imbalance. The main drawback of using global barriers is the load imbalance caused by the diagonal-wide inter-tile synchronization. The imbalance is caused due to the varying number of tiles in each diagonal and the count is not always a multiple of an integer, hence all SMs need to wait until the SMs with the highest number of assigned tiles finish. We present an example in Figure 2 that shows the execution time line of a wavefront parallelism to demonstrate the load imbalance caused by global barriers. Figure 2(a) shows a 2D data partitioned into 8 tiles in each dimension and executed on 4 SMs using global barriers. Tiles in a diagonal are shaded with same color and numbered with the SM ids that are assigned to execute them. The scheduling is assumed to be the default round-robin TB scheduler believed to be used by NVIDIA GPUs. Figure 2(b) shows the time line for each SM where shaded rectangles correspond to busy time and dark-black slots correspond to idleness. Due to varying length of tile diagonals, some SMs are left idle for nearly all the diagonals. Figure 2(c) shows the ideal execution if global barriers are removed and SMs synchronize directly with each other to enforce dependencies. As we can see, the idleness due to inter-tile load imbalance is completely removed. Theoretically the saved time can be as high as 25% based on the total tile and SM count.

$$idle(d,p) = \begin{cases} p - |d| mod p, & \text{if } |d| > p \\ 0, & \text{otherwise} \end{cases}$$

$$n_{idle} = \sum_{d=1}^{t_x+t_y-1} idle(d,p)$$

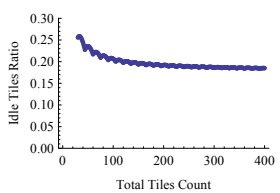

Figure 3: Number of idle tile execution slots on the left(a) and the ratio of idle slots to total tiles on the right(b).

The number of idle tile execution slots can be expressed with the equation given in Figure 3(a), where p is the number of processors and d is the diagonal iteration index. Using this equation, we have plotted the ratio of idle tile slots to the total tile count, which increases as smaller tiles are being used. As shown in Figure 3(b), even using smaller tiles to increase total tile count does not help this imbalance to be eliminated.

The rest of the paper describes PeerWave that provides the ideal execution of the rightmost Figure 2(c).

3. PEERWAVE

We propose PeerWave to employ direct SM-to-SM synchronization instead of global synchronization for wavefront parallelism to address the above drawbacks. As shown in Figure 4, PeerWave assigns a row of tiles to an SM to improve inter-tile locality and it uses decentralized synchronization to avoid barriers and load imbalance, where synchronization points are appropriately placed to maximize performance. Light triangles on the bottom-right of each tile correspond to the points where an SM writes to neighboring SM's flags after it finishes processing the tile; dark triangles on top-left of the tiles are for the places where a dependent SM reads these flags to continue execution. The rest of this section describes PeerWave in detail.

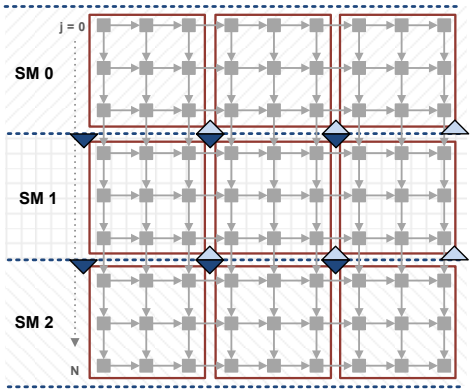

Figure 4: PeerWave: Partitioning of 2D iteration space into rows of tiles and assignment of rows to SMs. Triangles correspond to synchronization points, (light=write, dark=read).

3.1 Row-to-SM Assignment

PeerWave assigns a **tile-row**, a complete horizontal row of tiles in the 2D iteration space, to a single SM to exploit locality across tiles. In case there are more rows than SMs, the rows are distributed among the SMs in a round-robin fashion. Since CUDA does not provide any native mechanisms to manually assign thread blocks (TBs) to SMs, a TB is created for each row and *the TB-to-SM affinity* is achieved by limiting total number of TBs in a kernel launch to total number of SMs.

Row-to-SM assignment exploits locality by allowing elements close the horizontal border between tiles to remain in caches or local memory until they are accessed again. When combined with hyper-tiling, *tile-row*s further increase reuse across multiple columns with more hits on the last level cache. Moreover, this approach allows uninterrupted processing of elements across different tiles and enables two further optimizations, use of shared memory diagonal buffers and hybrid row-diagonal data storage (described in Subsection 3.4).

Algorithm 2 PeerWave main loop

```
1: for every row r assigned to SM i do
2:     for every tile t in row r do
3:         wait until flag[t, i] = 1
4:         process_tile(t, r, i)                ▷ SIMD
5:         flag[t, i + 1] ← 1
6:     end for
7: end for
```

Algorithm 2 shows how each SM processes its tile-rows. The outer loop iterates over the rows assigned to an SM in a round-robin fashion whereas the inner loop processes the tiles in a given row in sequence. Each SM processes its tiles one at a time while performing the major computation, i.e. processing of a tile, in parallel using SIMD computation units. In the inner loop, right before and after processing of a tile, the SM communicates with its neighboring SMs, called *Peer-SM*s, to enforce the dependencies between the tiles in adjacent rows.

3.2 Peer-SM Synchronization

Peer-SM synchronization is the key component of our method that replaces the global barrier synchronization between diagonal tiles with distributed synchronization among neighboring tiles. The decentralized nature of Peer-SM allows SMs within a GPU to work independently through synchronization points similar to CPU multithreading. Each SM processes the tiles in its row in dependence order and communicates with the SMs assigned to preceding and succeeding rows before and after processing each tile. By avoiding the redundant waits on global barriers and relying only on direct communication between neighboring SMs, Peer-SM improves load balance.

We implement inter-SM communication by using one-way *flags* located in the GPU global memory. Each SM owns a dedicated array of these flags corresponding to the tiles assigned to it. The one-way communication pattern is realized in PeerWave by having an SM read flags only from the prior row and write only to the flags for the succeeding row. Unlike the previous global-barrier based approaches, this one-way communication removes the need for atomic operations and locks, and scales well with increasing number of SMs.

When an SM is ready to process a tile, it continuously polls the corresponding flag, and begins processing once the flag is set to 1. After the SM has finished processing the tile, it sets the flag for the tile in the next SM's row that is waiting on the current tile, and then becomes ready to process the next tile in the row. When there are more rows than the number of SMs, and multiple rows are assigned to the SM, the SM resets all synchronization flags and starts processing next row immediately after the last tile of the current row has been completed. Per-SM flag arrays are created only once and then reused as the SM moves from one row to its next assigned row. Implementation details of Peer-SM synchronization are given in Section 5.

3.3 Flexible Hyper-Tiles

While tile-row assignment and peer-SM synchronization improve inter-SM load balancing with better data locality, further speedup can be achieved during intra-SM computation. Rectangular tiles result in shorter diagonals at the beginning

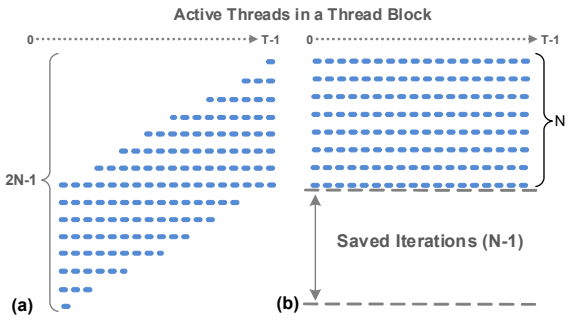

Figure 5: Intra-tile thread utilization is demonstrated for rectangular tiles on the left(a) and for hyperplane transformed tiles on the right(b)

and ending iterations within a tile and this partitioning leaves many threads (hence CUDA cores) in a TB idle. Hyperplane tile transformation improves SM utilization by maintaining a uniform diagonal size across the entire tile and thus maximizing the throughput. The number of iterations needed for intra-tile computation for square and hyperplane tiles are shown in Figure 5(a) and (b), respectively. The hyperplane tile partitioning decreases total number iterations from $2n-1$ to n in comparison to rectangular tiles. Thus, PeerWave uses hyperplane tiles (*hyper-tiles*) in addition to row-SM allocation and peer-SM synchronization. Each element in a diagonal is mapped to a thread in the TB and we iterate over diagonals until the end of the tile where inter-SM synchronization point is located. Figure 6(a) shows transformed tiles along with the synchronization points. Skewed indices change total number of tiles and shift the locations of inter-SM synchronization points. It should be noted that while [5] uses compiler level hyperplane loop transformations, unlike PeerWave, it still uses global barriers between diagonals.

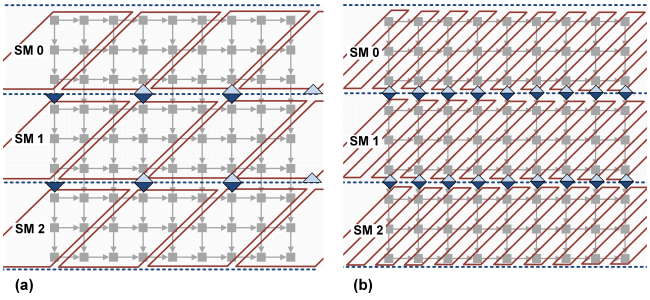

Figure 6: Hyper-Tiles on the left (a) and Flexible HT with variable synchronization intervals on the right (b)

We present *Flexible Hyper-Tiles* (FHT), an improved hyperplane transformation technique which allows us to *find* and *utilize* the optimum tile widths without changing the tile heights. Unlike the naive hyper-tiles approach with equal dimensions, FHT enables fine granular tile width adjustment for better control over the frequency of synchronization while maximizing intra-SM utilization without requiring tile heights to be changed. Smaller synchronization intervals decrease the time SMs must wait before processing their rows during the initial and final phases of execution when the parallelism is limited. Figure 6(b) shows an example of tile resizing where SM 1 can start processing tile (i=1,j=0) as soon as the diagonal corresponding to the element (i=0,j=2) has been processed by SM 0.

Different from prior studies, *Flexible Hyper-Tiles* uniquely models the optimal synchronization intervals (i.e. tile width) and finds the best trade-off between decreased load imbalance and increased synchronization cost. Optimal intervals are application and device specific and we develop an analytical model that outputs best tile width for given application/device characteristics. PeerWave runtime auto-adjusts the synchronization points based on the optimal tile width obtained from this model. Derivation of the model is presented in Section 4.

3.4 Optimizations

We further improve performance of PeerWave by employing the following optimizations.

Shared memory diagonal buffers: An important feature of PeerWave is its uninterrupted sequential processing of tiles within a row on the same SM. This enables efficient use of SM shared memory between consecutive iterations of tiles to dramatically reduce global memory accesses. Since processing a diagonal requires access to elements from preceding and succeeding diagonals, corresponding data can be stored in a sliding window of *diagonal buffers* that eliminates repetitive accesses to global memory as the diagonals are iterated upon by the same SM. When a diagonal is no longer to be accessed, it is written back to the device memory. The size of the sliding window is dependent on the length of the longest dependency vector in the transformed iteration space. FHT greatly increases the efficiency of diagonal buffers by combining accesses to partial diagonals, which are located towards the edges of neighboring tiles, into consecutively accessed 'full' diagonals.

Hybrid row/diagonal-major data layout: The diagonal access pattern during intra-tile processing requires CUDA cores inside SM to access non-consequent locations in the memory. This pattern results in highly un-coalesced memory accesses and causes significant slow down due to excessive number of memory transactions. Diagonal-major data representation [2] improves coalescing by storing elements in the same diagonal in consecutive memory locations. However, this approach does not preserve inter-tile locality, and causes the border elements to be redundantly being read and written as the processing of diagonals in one tile finishes and the next one begins. To prevent this behavior, we use a hybrid approach where the tiles in a row are continuously placed in the global memory in row-major order while the elements in the same diagonal are placed in diagonal-major order. The hybrid row-diagonal major data layout does not cause any memory space to be wasted since all diagonals in hyper-tiles have the same length.

The change in storage format requires the host data to be initialized properly. This can be done in two ways: (1) The data is converted to/from hybrid layout before and after GPU memory transfers. (2) When initializing the data from I/O resources or dynamically, the index transformation can be performed on the fly. The former method requires additional host-to-host data copy operations, therefore can be costly. The latter solution eliminates redundant copies and requires only a few additional instructions per index, whose cost are already hidden by much longer I/O instructions. We provide a simple API, whose details are given in the next section, to translate given x, y coordinates to proper data indices.

4. FINDING OPTIMAL TILE WIDTH

Next we develop an analytical model for finding the optimal tile width that will be utilized by Flexible Hyper-Tiles(FHT) to minimize the total execution time, which is a function of diagonal execution time and peer-SM synchronization cost. We compute the total execution time by identifying the critical path, i.e. the longest path through the execution. The output of our model is the optimal tile width for use in the runtime developed later.

Symbol	Definition
P	Number of SMs
W / H	2D input data width / height
w_t / h_t	Tile width / height
d	Diagonal execution time (measured)
τ_s	Peer synchronization cost (measured)
τ_e	Tile execution time

Table 1: The notation used by our model

The notation used in the derivation of the model is given in Table 2. We assume that the input is a 2D data matrix with height H and width W. The matrix is partitioned into H/h_t tile rows and P SMs are assigned to these rows in a round-robin fashion. The constants τ_s and d represent peer synchronization cost and the processing time of each intra-tile diagonal, respectively. These parameters are obtained using the *findOptimumTileWidth()* function which is provided by our API and performs an offline run on a small fraction of the input data to measure these parameters. Tile execution time τ_e is derived as a function of d, where:

$$\tau_e = w_t d \tag{1}$$

We define B to represent the time during which the processing of a row assigned to the first SM is blocked by the execution of the previous row assigned to the same SM. This case happens when there are fewer SMs than the number of tile columns. B is defined as follows:

$$B = \frac{W + h_t}{w_t}(\tau_e + \tau_s) - P(\tau_e + \tau_s) \tag{2}$$

The critical path consists of processing the first two tiles of every row and all the tiles in the last row. The point where blocking (B) occurs, marked by a dashed arrow, is where the first SM finishes processing the last tile in a row and starts the first tile in its next assigned row. Hyperplane partitioning introduces an extra tile at the beginning of each row as there is no tile on which the first tile in a row is dependent.

Using equations 1 and 2, we can formulate the critical path, hence *the total execution time*, as follows:

$$\begin{aligned} T_{total} = &\frac{H}{h_t}\left(\frac{h_t}{w_t} + 1\right)(\tau_e + \tau_s) + B\left(\frac{H}{h_t P} - 1\right) \\ &+ \left(\frac{W}{w_t} - 1\right)(\tau_e + \tau_s) \end{aligned} \tag{3}$$

According to Equation 1, enlarging h_t is always beneficial as this will lead to better intra-tile utilization and tile execution time will not increase because τ_e is invariant w.r.t h_t. On the other hand, any values w_t, s.t. $h_t \geq w_t$, will not change tile execution time. However, by analyzing equation 3 (details omitted), it can be shown that total execution time decreases as h_t increases. Thus, using maximum h_t (i.e., maximum number of threads per intra-tile diagonal) for hyper-tiles will always give best performance. With fixed h_t, total execution time becomes a function of w_t, the synchronization interval.

Equation 3 can be transformed into the form $aw_t + b/w_t + c$, where the optimal w_t that minimizes the equation is given by $\sqrt{b/a}$. Hence, the optimal synchronization interval based on the model is given by:

$$w_t = \sqrt{\frac{\tau_s(HW + Hh_t + Hh_t P - h_t^2 P)}{dh_t P(P-1)}} \tag{4}$$

We use equation 4 to find the optimal synchronization interval without the need for running the application on the entire input data. We validate the above model in the Evaluation Section.

5. RUNTIME & API

We develop a runtime to handle inter-SM synchronization and an easy to use API for users to implement wavefront applications.

PeerWave runtime employs a multi-core like execution scheme with each SM acting as an independent SIMD-capable processor. It uses persistent TBs each of which runs until all corresponding tiles are processed. The run-time kernel is launched with TB count set equal to the total number of physical SMs in the GPU. The default NVIDIA TB scheduler assigns each SM in a round robin fashion, and the number of TBs per SM is determined by the resource requirements of the kernel. For this reason, in our runtime, we use large enough TBs (hence tiles) so that HW scheduler only assigns one TB per SM. A major benefit of having persistent TBs is that SMs can preserve execution state in their local memory (e.g., shared memory in CUDA) and also retain application data across iterations.

```
struct user_data{float* data; long w; long h;
    int** dependencyVectors;}
void transformIndex(int& x, int& y, int& index,
    int tile_w, int tile_h, user_data udata);
long findOptimumTileWidth(user_data udata,
    int tile_h,int nSM,float trainingRatio);
__global__ void PeerWaveKernel(user_data udata,
    int tile_w, int tile_h, int nSM, int* flags);
```

Listing 1: PeerWave kernel header and other helper functions

PeerWave runtime consists of a user-modifiable data structure, preparation functions, a main runtime kernel, and peer-SM communication functions called by this kernel. The headers for these components are given in Listing 1 and the details are elaborated in the rest of this section.

User data: *user_data* struct contains device memory pointer to the user data, dimensions of the 2D data space and the *dependencyVectors* for the stencil computation. Users are responsible for initializing the device memory via *cudaMalloc()* calls and copy initial data from host memory to GPU via *cudaMemcpy()*. *user_data* struct can be modified to add application specific pointers and parameters.

Preparation functions: We present users two optional pre-runtime functions to increase the usability of our framework. The first one is the index transformation function, *transformIndex()*, to allow users a convenient two-way index translation from/to 2D x and y coordinates to/from a data *index*. The function uses the application parameters given in *user_data* and performs the transformation to support our row/diagonal major hybrid data layout. The second function

is *findOptimumTileWidth()* and it is used to find optimum tile width via the provided model. The function finds applicationarchitecture specific parameters required by the analytical model by performing an offline training run on a fraction of the input data specified by the *trainingRatio* parameter.

***PeerWave* Kernel:** PeerWave runtime, hence the user algorithm, is initiated by launching this kernel with the following parameters:

- *TB Size & Grid Size*: These two parameters are supplied to CUDA runtime while launching the kernel. The grid size (i.e., the number of TBs) is set equal to the number of SMs (nSM) whereas TB Size is application and architecture specific and the programmer need to determine the optimal size according to resource requirements of the wavefront application.
- *nSM*: GPU specific value denoting the total number of SMs.
- *Flags*: The communication flags which need to be preallocated on GPU memory via provided *init()* function. The size of this array is dependent on the number of SMs and the number of tiles in a row.
- *tile_h*: Height of the tiles. Since PeerWave maps each element in the vertical axis of a tile to a thread in the TB, *tile height* should be equal to the TB Size (i.e. number of threads) specified during kernel launch.
- *tile_w*: Width of the tiles (i.e. synchronization interval). As discussed in Section 3 and 4, tile width determines the synchronization interval and it is an important parameter that directly affects the performance. Users can either manually derive the optimal tile width using Equation 4 derived in Section 4 or use *transformIndex()* as described above.

```
flagRead(){
  if (threadIdx.x==0){ // Single thread only
    // check tile flag until set
    while(flags[(nTiles_x)*smID+j]==0)
      cosineSleep(); // dummy computation
    flags[(nTiles_x)*smID+j]=0;} // reset for
   future
  syncthreads(); // sync with other threads}

flagWrite(){
  if (threadIdx.x==0){ // Single thread only
    int nextSM = (smID+1)\%nSM;
    // Set flags for next row.
    flags[(nTiles_x)*nextSM+j]=1;}
  syncthreads();} // sync with other threads
```

Listing 2: Implementation of flag read (a) and write (b) operations

Once the *PeerWaveKernel* is launched, PeerWave main loop given in 2 is executed by each SM until all the data is processed. The main loop involves read of synchronization *flags*, processing of the tile and writing back to the *flags*.

Flag Read: PeerWave relies on efficient synchronization of SMs via one-way communication flags. Listing 2(a) shows the code used to ensure that the corresponding flag is set (i.e., dependencies are met) before every SM starts processing a new tile. Since NVIDIA does not provide an efficient means of inter-SM communication, we keep polling the flag corresponding to the executing SM (smID) and the current column (j) in the tile row until its value is 1. We decrease potential memory contention by idling the SM for a small short period of time using *cosineSleep()* function derived from repetitive calls to cosine function. When the flag is detected as set, we reset it for future use and move on to tile processing.

Tile Processing: Each SM processes the elements in the same diagonal in parallel using the threads in the TB. Each row of elements is mapped to a single thread and the number of columns in the tile correspond to the number of diagonals. After processing of each diagonal, threads inside the SM synchronize using *cudaThreadSync()* method, which is an efficient intra-SM HW barrier implementation provided by CUDA.

Flag Write: Once a tile is processed, SMs need to set the flag for the next SM (smID+1) in the same column (j) as well as the tile in the next column(j+1) in the same row, as shown in Listing 2(b). Border conditions are omitted for simplicity. The overhead of both flag read/write operations are measured to be minimal and reported in the evaluation section.

Element Processing: PeerWave enables users to implement their core computation by overriding the *compute_element()* function as given in Listing 3. The Listing shows an example overriding of *compute_element()* function to implement the core computation of Smith-Waterman algorithm. During tile processing, runtime performs hyper-plane transformations on border elements of each tile and calls user function with the base tile index as well as the element's global x/y coordinates. *u_data.seq1* and *u_data.seq2* are the two input data sequences added to the *user_data* struct and the *u_data.data* array corresponds to the intermediate 2D computation matrix used while calculating the similarity between the two sequences. *compute_element()* function is run in SIMD fashion where all threads in the TB are called with corresponding x and y coordinates of the elements that they are mapped to.

```
__device__ void compute_element(
    const struct user_data u_data,
    int tileRow, int tileColumn,
    int tile_w, int tile_h,
    int x, int y, int index, int nTiles){
  int upleft,left,up;
  if (u_data.seq2[y] != u_data.seq1[x])
    upleft = u_data.data[index-2*tile_height-1]-1;
  else
    upleft = u_data.data[index-2*tile_height-1]+2;
  left = u_data.data[index+tile_height]-1;
  up = u_data.data[index-tile_height-1]-1;

  int max = MAX(0,MAX(upleft, MAX(left, up)));
  udata.data[index] = max;}
```

Listing 3: An example override of *compute_element()* function.

6. WAVEFRONT APPLICATIONS

We evaluate PeerWave using 6 different wavefront parallelizable applications.

Heat 2D simulates dissipation of heat over a two dimensional surface during a specific time frame. During each simulation step, the algorithm calculates the heat of every point in the 2D surface by averaging the heats of its neighboring elements. The calculation relies on the values calculated in current and previous simulation step as follows:

$A[i,j] = (A[i-1,j]+A[i,j-1]+A[i+1,j]"+A[i,j+1]")/4$

SOR loops are generalized form of time based stencil computations on neighboring elements. We have considered a 2-dimensional SOR loop iteration with 5-point stencil which is similar to Heat2D but with including the self element in the average as well. The core computation is as the following:

$$A[i,j] = (A[i-1,j]+A[i,j-1]+A[i,j]" \\ +A[i+1,j]"+A[i,j+1]")/5$$

Smith Waterman (SW) is a well known algorithm used for local sequence alignment. It takes two sequences as inputs and builds a matrix to mark and extract matching patterns. The algorithm steps backwards by one position in the three directions (i-1,j), (i, j-1), and (i-1, j-1). The related computation is shown below, where w is the penalty function that returns 2 on equality and -1 otherwise:

$$A[i,j] = \text{MAX}(A[i-1,j]-1, A[i,j-1]-1, A[i,j+1]+w(i,j))$$

Dynamic Time Warping (DTW) is used to find an optimal match between two given sequences (e.g., time series). The series are "warped" in the time dimension to find a measure of their similarity without relying on certain non-linear variations in the time dimension. The pseudo code showing cost calculation across neighbors is as the following:

$$A[i,j] = |i-j|+\text{MIN}(A[i-1,j]+A[i,j-1]+A[i-1,j-1])$$

Summed Area Table (SAT), also known as integral image, is used for generating the sum of values in a rectangular subset of grid. Summed area table can be computed efficiently in a single pass over the image using the equation below:

$$A[i,j] = w(i,j)+A[i-1,j]+A[i,j-1]-A[i-1,j-1])$$

Integral Histogram (INT), is another popular image processing algorithm to calculate histogram of a given bitmap. Different from SAT, it reads from the original image (i.e. bin) but writes into a separate data location used to store corresponding histogram as shown in the algorithm below:

$$A[i,j] = B[i,j]+A[i-1,j]+A[i,j-1]-A[i-1,j-1])$$

The dependencies required by above applications are illustrated in Figure 7. As shown on the left hand side of the figure, Heat2D and SOR computation on each element *(i,j)* depends on the top *(i,j-1)* and left *(i-1,j)* elements which are calculated in the previous wave, as well as bottom *(i,j+1)* and right *(i+1,j)* elements that are calculated in the previous time step (denoted with ") of the algorithm. Other four algorithms, SW, SAR, DTW and INT, have similar dependencies. Different from Heat2D and SOR, their computation also relies on the top-left element *(i-1,j-1)*.

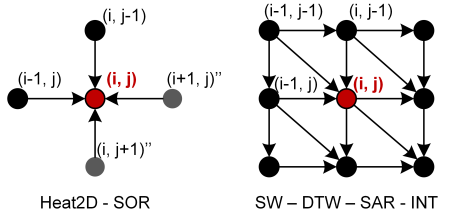

Figure 7: Computation dependencies required by Heat2D, 2D-SOR, SW, DTW, SAT and INT.

7. EVALUATION

In this section we first describe the details of our experimental setup and then present the experimental results.

Architecture. We evaluate PeerWave on NVIDIA's Tesla K40c series GPU attached to an AMD Opteron 6100 based system. The GPU has 15 SMX units each having 192 CUDA cores accessing 12GB of global DDR3 memory. K40c has a shared L2 cache size of 1.5MB and shared memory of each SMX is configured to use 16KB/48KB L1 Cache/Shared with global memory requests are not set to be cached on L1 (Kepler's default setting).

Methodology. In our evaluation, we only measure GPU kernel computation times only because our proposed technique focuses on kernel execution to improve overall execution. For all experiments, we use hybrid row/diagonal major data layout with on-the-fly index transformation as described in Section 3. Since data initialization and transfer times are observed to be similar w/ and w/o the hybrid layout, we exclude all host related operations (initialization and transfers) from our experiments to demonstrate the efficiency of our technique in more detail. Overlapping of data transfers and kernel computations for further speedup is orthogonal to our proposed method and not covered in this paper.

In all of our experiments, we fix the data size to the highest amount that fits in GPU's global memory, and set the tile height and TB size to 1024 (unless specified otherwise in the experiment). For optimal performance, we implement global barriers by using NVIDIA Kepler's dynamic parallelism that allows in-GPU *cudaDeviceSync()* calls. We compare the following wavefront execution methods:

1. *Global barriers*: The baseline global synchronization approach using square tiles.
2. *PeerWave*: Basic implementation of our proposed peer-SM synchronization based technique using square tiles.
3. *Global barriers w/ HTiles*: Global synchronization approach using hyperplane transformed tiles with equal dimensions.
4. *PeerWave w/ HTiles*: Improved version of PeerWave using hyper-tiles with equal tile dimensions.
5. *PeerWave w/ Flexible HTiles*: Optimized version of PeerWave hyper-tiles with flexible non-equal tile dimensions.

7.1 Execution Time: Equal Tile Dimensions

We first evaluate the performance of the first four techniques all of which use equal tile dimensions. Figure 8 shows execution times of each application run for varying tile sizes. TB size is set as 1024, all 15 SMs in the GPU are used and tile width and height are varied together between 256 and 1024. The first two bars correspond to rectangular (square) tiles and the last two correspond to hyper-tiles. The second and fourth bars represent improvements due to PeerWave technique.

An immediate observation is the dramatic decrease in execution time with the use of hyper-tiles. This is due to the decrease in the number of intra-tile diagonal iterations from $2n-1$ to n. PeerWave further improves the performance in both hyper-tile and square tile cases due to increased inter-tile locality and decreased SM idle times. PeerWave shows better performance when using hyper-tiles due to decreased memory contention. The details of the speedup will be analyzed throughout the remainder of this section.

The results also show that the decrease in tile sizes reduces the performance due to under-utilization of CUDA cores inside each SM. Since the maximum intra-tile parallelism is limited by the minimum dimension in the first four execution methods, tile size of 1024x1024 (hence TB of size 1024) gives maximum performance.

7.2 Execution Time: Flexible Hyper-Tiles

Flexible hyper-tiles (method 5) decreases total idle time by increasing number of synchronization points via smaller tile widths. Also, intra-tile utilization is kept maximum by using largest tile heights. The optimal tile width is determined by the provided model and then passed to PeerWave kernel as a

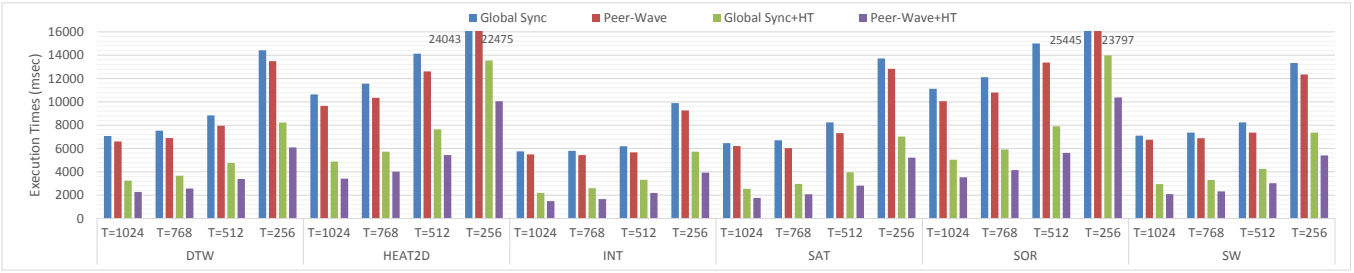

Figure 8: Kernel execution times for global barriers and PeerWave approaches with and without the use of hyper-tiles.

parameter. In this subsection, we first evaluate the accuracy of our model in finding the best synchronization intervals, and then using the optimal values, we compare Flexible hyper-tiles (FHT) method with the best performing equal tile dimension methods 3 and 4.

7.2.1 *Model Verification*

To verify the accuracy of tile width optimization model presented in Section 4, we have compared the execution time estimated by Equation 1 with the measured execution time for the FHT method. The model parameters, d and τ_s are measured in an offline run using a 0.6% of the total data. This ratio corresponds to the initial corner region which spans 135 tiles out of 2025 in total. We fix the tile height to 1024 and vary the tile width from 2 to 1024. The comparison between the estimated and measured execution times is shown in Figure 9.

Overall, our model represents the execution pattern of PeerWave approach quite well. It can be seen that both experiment and simulation curves have a U shaped pattern which points to the trade-off between synchronization overhead and idleness reduction. The slight discrepancy between the simulation and experiment is due to variation of the values d and τ_s throughout the execution. Since we are using only an initial portion of the execution, the increased memory contention towards the middle region of the input matrix affects the blocking time B and hence the critical path. Although the execution times are slightly different, the synchronization interval estimation is accurate.

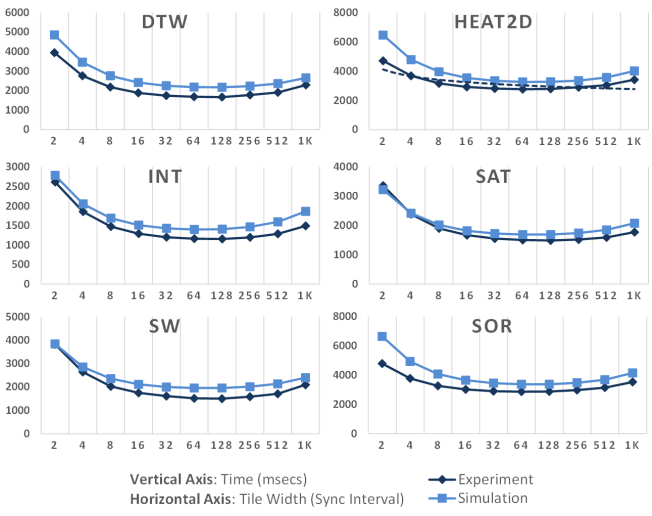

Figure 9: Comparison of numerical simulation and experimental results for Flexible hyper-tiles.

7.2.2 *Flexible Hyper-Tiles (FHT)*

To evaluate flexible hyper-tiles, we take method 3, global synchronization with hyper-tiles (GS+HT), as our baseline and compare its performance with methods 4 and 5, i.e. PeerWave w/ hyper-tiles (PW+HT) and PeerWave w/ Flexible hyper-tiles (PW+FHT). We fix both tile height and width to 1024 for GS+HT and PW+HT, as these give the best results. On the other hand, for PW+FHT, we have set tile height as 1024 and varied tile width between 64, 128, and 256, which are the most close-to-optimal values reported by our model as well as the experiments.

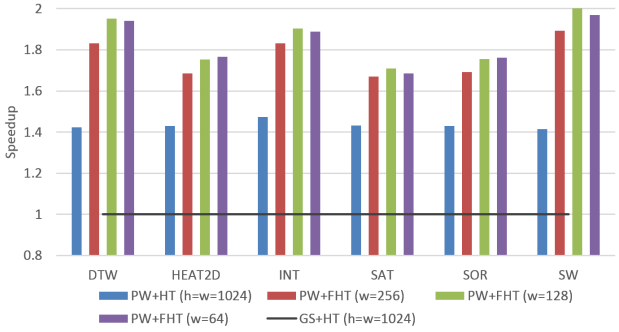

Figure 10: The speedup obtained by Flexible hyper-tiles method.

As the results given in Figure 10 show, flexible hyper-tiles approach improved upon our original PW+HT technique and increased the final speedup by up to 2x compared to the recent approach based on global barriers with hyper-tiles (GS+HT) [5]. When compared to the effects of using smaller tiles with equal dimensions shown in Figure 8, the results clearly show the benefits of changing synchronization interval without affecting utilization.

7.3 Idle Times

To further analyze the effects of peer-SM synchronization we measure the average time that an SM stays idle and compare it to the idle time spent on global barriers. As previously shown in the Figure 2 in Section 2, we classify the idle time into two categories: (a) *Corner*, the compulsory wait time spent during the beginning & end of the execution, where the parallelism is less than total number of SMs(15); and (b) *Intermediate*, where there are always more parallel tiles than SMs.

The results in Figure 11 shows the break down between border (blue) and intermediate (orange) idle times for each application and the technique. PeerWave dramatically reduces the *intermediate* idle time both with and without hyper-tiles,

whereas the *corner* idle times remain similar as expected, due to limited parallelism in those regions.

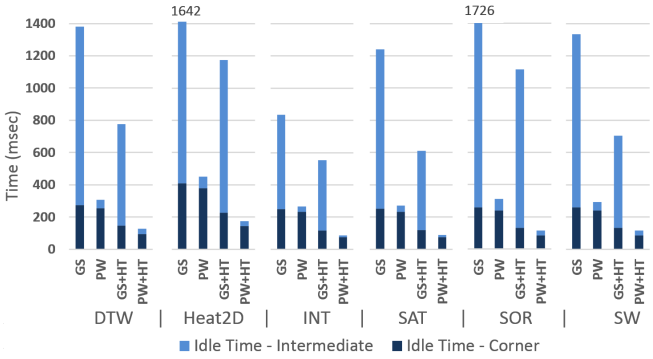

Figure 11: Breakdown of Per-SM average idle time between two phases of the execution.

Reduced idle times cause more SMs to be active hence result in increased memory throughput as shown in Figure 13. PeerWave was able to improve memory bus utilization more in both cases with and without hyper-tiles. On the other hand, use of hyper-tiles dramatically reduces total memory transactions due to reduced number of intra-tile diagonal iterations and increased coalesced accesses, therefore less throughput is needed.

7.4 Tile Locality

Another important drawback of global barriers is overcome by PeerWave via increased inter-tile locality due to row-SM assignment. To observe this, we measure average time required to execute a single tile and give the results in Figure 12(a). Tile execution times are reduced between 15% and 25% compared to global barriers.

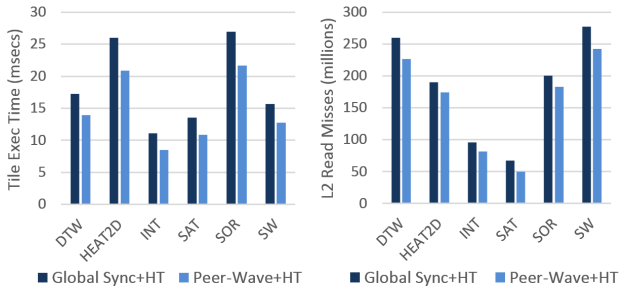

Figure 12: On the left (a) average per-tile execution times and on the right (b) total L2 read misses

We have also profiled the total number of L2 read misses using NVIDIA's profiling utility, **nvprof**, and report the results in Figure 12(b) for 32-byte transactions. As expected, PeerWave reduces the misses in both cases. Although the improvement seems to be minor, the effect on execution time is considerable due to high cost of a global memory access.

7.5 Synchronization Cost

We measured the total time spent during *flagRead()*, *flagWrite()* and global barriers using the device-side clock64() CUDA function. The cost of updating global barrier mutexes in global-synchronization with HT (GS+HT) and the communication flags in PeerWave with HT (PW+HT) is given in milliseconds in Table 2. The values correspond to the total

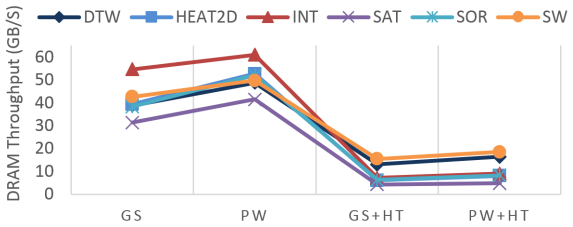

Figure 13: Total GPU global memory throughput.

time spent for synchronization . Although PeerWave improves the synchronization times by around 25%, the total sync cost in both cases is actually marginal and only corresponds to under 0.1% of the total execution times. Most of the speedup obtained by PeerWave is achieved via improved load balance and locality.

	DTW	HEAT2D	INT	SAT	SOR	SW
GS+HT	4.28	4.25	3.77	4.25	4.27	4.25
PW+HT	3.49	3.39	2.37	3.36	3.4	3.43

Table 2: SM Synchronization times in milliseconds.

8. RELATED WORK

Wavefront parallelism in the literature has drawn attention under various contexts like dynamic programming, stencil computations and time based iterations (i.e. Gauss-Seidel or Jacobi) and we discuss the GPU related sub-set of the literature in this section.

Datta et.al [3] are among the first who study the effectiveness of stencil computation on GPUs and compare the performance with multi-core architectures. They focus on data allocation and bandwidth optimizations and present a run-time that auto-tunes architecture specific parameters. [18] is another early study which employs tiling and successive placement of data on the memory for coalesced access on GPUS. They implement in-kernel global barriers using device memory, to prevent going back to CPU for ensuring diagonal dependencies and compare the performance with host-based *cudaDeviceSync()* function. Yan et al. [19] employs synchronization between different stages of parallel prefix sum to remove global barriers between reduce and scan phases.

To decrease un-coalesced memory accesses and increase locality, the study in [2] proposes a CUDA based API, Dymaxion, that remaps memory layout via on-the-fly index transformations. They implement diagonal-strip mapping which assigns tiles in the same diagonal to consecutive memory locations. Sandes et al. [14] implements Smith-Waterman algorithm on GPUs via application specific optimizations like block pruning and multi-stage algorithm execution. As an attempt to improve inter-tile load balance, the study in [9] speculatively executes tiles of Gaussi-Seidel, Jacobi and SOR loops, and tests them for accuracy as the iteration asynchronously proceeds. However, eliminating global barriers removes the need for such techniques.

Polyhedral loop transformations are commonly used to improve intra-tile utilization. The study in [5] proposes compiler-level transformations for arbitrary loop nests. Loop iteration spaces are skewed using affine transformations where the transformation matrix is determined by the dependency vectors. Grosser et al. [6] uses trapezoidal tiles and automatically generates CUDA code to transform given loop nests. Another

paper [7] proposed a stencil compiler that facilitates inverted and upright split-tiling for improved parallelism. The study in [1] considers diamond tiling for time iterated computations with periodic data domains to improve parallelism and locality. However, all of these approaches still rely on global barriers between tile diagonals.

Several studies have also focused on heterogeneous execution on multi-core CPUs and multiple GPUs. Among the most notable ones, [20] automatically generates 3D stencil code for multi-GPU computing. A more recent study [12] partitions the load across multiple CPUs and GPUs while automatically adjusting tile sizes in the run-time for the best load balance across heterogeneous processors.

9. CONCLUSION

In this paper we introduced PeerWave, a new parallelization scheme for nested loops with data dependencies. Our approach eliminates inter-tile global barriers and uses SM-to-SM direct communication instead. We improve inter-tile load balance and locality that greatly reduces SM idle times. We further improve PeerWave with Flexible Hyper-Tiles which allows fine-granular synchronization interval adjustment while keeping intra-tile utilization at its maximum. We showed that PeerWave can achieve speedup of up to 2x when compared to existing global barrier based approaches.

10. REFERENCES

[1] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache. Tiling and optimizing time-iterated computations on periodic domains. In *Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT)*, pages 39–50. ACM, 2014.

[2] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of the 25th ACM international conference on Supercomputing (ICS)*, page 13, 2011.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (ICS)*, page 4. IEEE Press, 2008.

[4] P. Di, H. Wu, J. Xue, F. Wang, and C. Yang. Parallelizing sor for gpgpus using alternate loop tiling. *Parallel Computing*, 38(6):310–328, 2012.

[5] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP)*, pages 350–359. IEEE Computer Society, 2012.

[6] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for gpus: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GP-GPU)*, pages 24–31. ACM, 2013.

[7] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS)*, pages 13–24, 2013.

[8] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988.

[9] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 213–222. ACM, 2010.

[10] N. Manjikian and T. Abdelrahman. Exploiting wavefront parallelism on large-scale shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 12(3):259–271, 2001.

[11] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing (ICS)*, pages 256–265. ACM, 2009.

[12] S. Mohanty and M. Cole. Autotuning wavefront applications for multicore multi-gpu hybrid architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, page 1. ACM, 2014.

[13] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 24th ACM international conference on Supercomputing (ICS)*, pages 1–13, 2010.

[14] E. F. d. O. Sandes and A. C. M. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(5):1009–1021, 2013.

[15] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, pages 365–376. ACM, 2012.

[16] M. Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[17] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing (SC)*, pages 655–664. ACM, 1989.

[18] S. Xiao, A. M. Aji, and W.-c. Feng. On the robust mapping of dynamic programming onto a graphics processing unit. In *15th International Conference on Parallel and Distributed Systems (ICPADS), 2009*, pages 26–33. IEEE, 2009.

[19] S. Yan, G. Long, and Y. Zhang. Streamscan: fast scan algorithms for gpus without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 48, pages 229–238. ACM, 2013.

[20] Y. Zhang and F. Mueller. Autogeneration and autotuning of 3d stencil codes on homogeneous and heterogeneous gpu clusters. *IEEE Transactions on Parallel and Distributed Systems (IPDPS)*, 24(3):417–427, 2013.